COMPUTER SCIENCE TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

UMIACS-TR-90-23
CS-TR-2410

February 1990

Knowledge Representation in PARKA*

Lee Spector†, James A. Hendler, and Matthew P. Evett†
Institute for Advanced Computer Studies and
Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

This paper describes a frame-based knowledge representation system called PARKA which runs on the Connection Machine. The PARKA language was designed to take advantage of the Connection Machine architecture in order to provide extremely fast query processing capabilities. While the primary goal in PARKA's design was to explore the use of massive parallelism in symbolic knowledge representation, we have also tried to respond to many of the criticisms and to take advantage of many of the advances documented in the literature of frame languages. PARKA provides a principled approach to multiple inheritance based on the work of Touretzky [21], and a new mechanism for the representation of part/whole relations. In addition, we have made efforts to make the intended *meanings* of PARKA representations clear from the perspective of work in the philosophy of language. The result is practical representation system that can run very quickly on parallel hardware. A specification of the syntax of the PARKA language is provided as an appendix.

*This work was partially supported by NSF grant IRI-8907890 and ONR grant N00014-88-K-0560. Additional support was provided by the University of Maryland Systems Research Center.

†Department of Computer Science, University of Maryland, College Park, MD 20742.

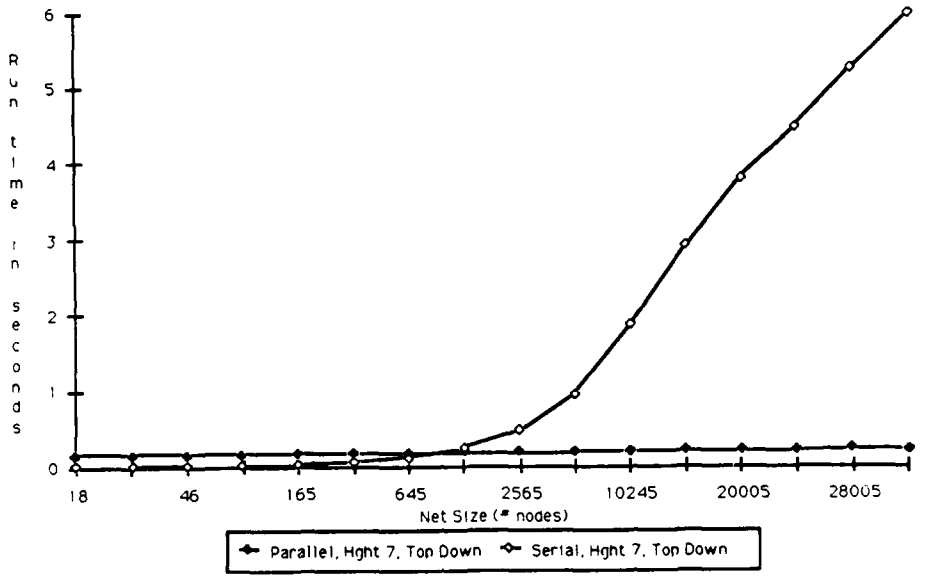

Figure 1: A comparison of parallel and serial runtimes for top-down inheritance queries.

1 Introduction

PARKA is a frame-based knowledge representation system with a massively parallel implementation on a Connection Machine. The PARKA language was designed to take advantage of the Connection Machine architecture in order to provide extremely fast query processing capabilities. The initial PARKA implementation was semantically simple but displayed dramatic speed improvements over its serial version. For example, simple inheritance queries (using a breadth-first ancestor ordering) required an amount of time proportional only to the *depth* of the knowledge representation hierarchy. Queries on even our largest (32000 frame) knowledge bases generally required less than 1 second to compute. A comparison of the serial and parallel run times for queries in randomly generated knowledge bases appears in Figure 1. The graph shows timings of the parallel implementation and those of a *stripped down* version of the serial implementation. The full serial version, which includes all of PARKA's features, type checking, etc., runs approximately 10 times slower. (For a discussion of these and related results see [6].)

Encouraged by the speed of our first Connection Machine implementation, we set out to make PARKA into a semantically well-grounded and useful knowledge representation tool. Implementation constraints prevented us from adopting pre-existing formalisms wholesale, or from adding "bells and whistles" without careful consideration. But while the primary goal in PARKA's design was to explore the use of massive parallelism in frame systems, we wished also to respond to many of the criticisms and to take advantage of many of the advances documented in the literature of frame languages. In addition, we wanted to make the intended *meanings* of PARKA representations clear from the perspective of work in the philosophy of language. The result is the language specification given in this paper. A serial prototype version of the full PARKA language has been written in Common Lisp. The parallel version is currently being built as an extension of our first Connection Machine implementation.

PARKA represents the references of category and individual terms by specifying *descriptions* of the category members and individuals to which such terms apply. (PARKA currently has no *assertional* component, in the sense of [2].) Although the descriptive facilities are not so elaborate as those of several previous systems (e.g., KL-ONE [3]), great care was taken in the design of those constructs that *are* provided. For example, PARKA provides a principled approach to multiple inheritance based on the work of Touretzky [21], and a new mechanism for the representation of part/whole relations. We feel that PARKA's facilities will be useful for a wide range of AI applications. (See the Conclusions for a brief discussion of the differences between PARKA and previous frame systems.)

PARKA's speed derives from its ability to perform certain types of inferences using parallel message-passing techniques [6]. Most PARKA queries require an amount of processing time proportional to the depth (and independent of the breadth) of a knowledge representation hierarchy. Since AI hierarchies tend to be wide and shallow, this means that query processing time is essentially independent of knowledge base size. The price for this performance is that most structural information must be represented explicitly. This requirement precludes uses of indirection, implicit knowledge generated at query time, and procedural attachment found in other systems. We can achieve similar effects, however, through the use of *topology constraints* enforced at knowledge base update time. A PARKA topology constraint in-

teracts with the system's knowledge base update procedures in order to make some given relationship explicit. The existence of a constraint can cause the system to perform integrity checks (and to reject an update if a check is not satisfied) and sometimes to add or delete links, create new frames, etc. As a result, many of our knowledge base update procedures are relatively expensive. This is a tradeoff we are willing to make for the sake of high efficiency queries, particularly since we envision the system's applications as being query-intensive and update-rare.

2 Basic Structures

The primitive representations in a PARKA knowledge base (KB) are *frames*. (We use the term "frames" because our structures bear significant resemblance to the "frames" of previous systems (e.g., [14, 18]), although there are also significant differences.) Consistent with standard terminology, a PARKA frame consists of a set of named, typed pointers called *slots*. The target of a slot's pointer is referred to as that slot's *value*, and the value is said to *fill* the slot. A slot can either be filled with a symbol or (more usually) with a pointer to another frame.

Each frame has a distinguished slot called *name*, which must be filled with a unique symbol. A frame represents a description of the referent of the term (the symbol) that fills its name slot. In certain cases the description can also be thought of as "picking out the reference" (in the sense of [12]) of the given term, but we do *not* claim that the notion of reference can thereby be reduced to description.[1] Frames come in two basic varieties, *individuals* and *categories*. Category frames describe the referents of category terms such as **dog**, **color**, or **philosopher**. Individual frames describe the referents of names used to refer to specific individuals in the world, such as **Fido**, **cyan**, or **Aristotle**.

Our notion of a *category* is different from that of a *set*, *class*, or a KL-ONE *Generic Concept* [3]. Set-theoretic and classificatory concerns are, in PARKA, of secondary importance to the description of the *referents of the terms* which name the frames. This is an important distinction because it

[1] This is a technical issue in the philosophy of language with which the casual reader need not be concerned. Those interested should see [12]

is our position that such knowledge cannot be adequately (or at least not compactly) represented by sets of necessary or sufficient conditions. This view was motivated by the considerations raised in work on the philosophy of language (such as [12] and [17]) and contrasts markedly with the view taken in work on KL-ONE [3]. We do not wish to digress into a philosophical discussion in this descriptive paper, but it should be noted that half of our view - that the conditions given in category descriptions are not to be taken as *necessary* - is implicit in most languages that allow the specification of exceptions to property inheritance. The problems which result from taking this position too far (i.e., by not providing *any* facilities for specifying that parts of descriptions are necessary or sufficient) have been explored by Brachman in [4]. PARKA avoids these problems by means of other mechanisms discussed below.

We view the filling of a category's slot as the addition of a property attribution to a description of the referents of a term. Hence the slot-values of a category and its position in the isa hierarchy (see below) can be thought of as describing a "typical" member of the given category. However, since most slot values can be overridden the descriptions are not normally necessary conditions for category membership (see, however, "Definitional Slots" and "Restrictions" below). Since category descriptions are not assumed to be complete, we also do not hold that such descriptions are *sufficient* conditions for category membership. There *are some* cases wherein we want category descriptions to behave as class descriptions (defined via set-theoretic constructs) but this is the exception rather than the rule (see the section "Set-Constructor Categories").

Frames are connected to one another not only by slots, but also by structures called *isa links*. Isa links are used to represent category membership and category/sub-category relations. An isa link's *source* can be either a category or an individual, while its *destination* must be a category. Note that we use "isa" in cases similar to those wherein others would use "instance" - we represent category membership by an isa link from an individual to a category. The set of isa links in any PARKA KB must form a directed acyclic graph (i.e., multiple inheritance is permissible but inheritance cycles are not). The user may pose queries about the category relationships encoded in the isa hierarchy, but more frequently these relationships are used for slot value *inheritance*, as discussed in the next section.

As mentioned above, a slot has not only a value, but also a *type*, which must be either SIMPLE, DEFINITIONAL, or RESTRICTION. If a category's slot is *not* simple then its value is a constraint on the values that the category's isa descendants may have for the given slot. This is explained further in the sections "Definitional Slots" and "Category Valued Slots and Restrictions" below.

3 Inheritance by Inferential Distance Ordering

When a particular frame has been given a value for a particular slot then we say that that frame is *explicitly valued* for the slot. If a frame is *not* explicitly valued for a given slot then queries for the slot value may still return a value; the frame may *inherit* a value from an isa ancestor.

For example, we may have a frame **elephant** with a slot **color** that is filled with **grey**, and another frame **circus-elephant** with no explicit value for **color**. If there is an isa link from **circus-elephant** to **elephant** (and if, for the sake of simplicity, **circus-elephant** has no other isa parents) then a query for the **color** of **circus-elephant** would return **grey**.

Inheritance of slot values is straightforward when all of the frames in a system have at most one isa parent; a frame inherits from the "closest" isa ancestor that is explicitly valued for the slot in question. In the presence of multiple isa parents, however, situations can arise which require special treatment. Previous systems often behaved erratically or produced counter-intuitive results in cases wherein a frame received different values from two (or more) different isa ancestors. Various strategies are available for resolving such conflicts. Although some of the simplest approaches (such as specifying that the ancestors are to be searched 'depth first up to joins') are adequate, for example, for inheritance in Object Oriented Programming languages [20], these strategies generally fail to resolve the semantic problems associated with inheritance in knowledge representation hierarchies. Touretzky has provided a rigorous analysis of these problems, and has proposed a solution based on what he calls the 'Inferential Distance Ordering' [21]. Although other, more elaborate solutions have since been proposed (most no-

tably [19]), it is Touretzky's solution that has been implemented in PARKA as this formulation rectifies the inappropriate behavior of previous systems without committing to positions on tangential issues (such as how conflicting evidence from disparate sources ought to be reconciled). The interested reader is referred to Touretzky's work for a discussion of the problems ('redundancy' and 'ambiguity') and of the reasoning which lead to the Inferential Distance Ordering; we discuss here only the implications of that work for the behavior of PARKA.

As an example of Inferential Distance Ordering, consider a query for the value of slot S of frame F. If F is not explicitly valued for S then the value returned (if any) will be inherited from one of F's isa ancestors. Suppose that n of F's isa ancestors (G1, G2, ..., Gn) are explicitly valued for S. If we have two explicitly valued ancestors, Gi and Gj, such that Gi lies on an isa path from F to Gj, then we say that Gi's value *overrides* that of Gj. We want F to inherit a value for S that is *not* overridden by any other value. If there is more than one such value then the specification is ambiguous and we do not want S to inherit any value at all. We can compute the value to be inherited as follows:

1) Put G1, G2, ..., Gn into Ancestor-Set.
2) Remove from Ancestor-Set any frame which is an isa ancestor of some other frame in Ancestor-Set.
3) If exactly one frame remains in Ancestor-Set then return the value of slot S for that frame; otherwise, if all of the remaining frames have the same value for the slot S then return that value; otherwise, return no value.

This is not the algorithm that PARKA actually uses; it is provided only to illustrate functionality. PARKA's algorithm for computing inheritance runs in time proportional to the depth of the isa hierarchy and requires no preprocessing or 'collapsing' of the isa hierarchy. (See [7] for more detail on the implementation.)

4 Definitional Values

One problem with early frame systems concerns the difference between using a slot value to represent a *default* and using a slot value to represent a *definition* [4, 24]. PARKA implements a definition as a slot value that cannot be overridden by any isa descendants of the given frame. (A similar approach was taken in [5].) We represent this 'non-overridability' as a topology constraint, implemented by making the type of the slot in question "definitional".

Definitional slots can be used, for example, to distinguish the sense in which a quadrilateral's 'four-sided-ness' differs from an elephant's 'four-legged-ness'. The **num-legs** slot of **elephant** will be of type SIMPLE and hence may be overridden (for the sake, say, of Clyde the three-legged elephant). The **num-sides** slot of **quadrilateral** will be DEFINITIONAL and hence its value will hold for *all* isa descendants of **quadrilateral**. All PARKA KB updates are checked for violations of definitional slots. If violations are found then the update is refused and an error is signaled. The time added to KB updates is expected to be proportional to the depth of the isa hierarchy (see [7]). Query time is unaffected.

5 Category Valued Slots and Restrictions

Slots in PARKA can be filled with at most one frame, but that frame may be a category. In order to represent slots that have many values we have established the representational convention that a slot filled with a category is thought of as being 'filled' (in a non-technical sense and only for the sake of a certain class of queries) by *all* of the *individuals* that are isa descendants of the filler. For example, consider the **color** slot of the frame **zebra**. To represent the fact that a zebra is (typically) both black and white we can create a category frame **zebra-color** with two individual frames (**black** and **white**) as isa children. When we fill the **color** slot of **zebra** with **zebra-color** (as in Figure 2) we are asserting that all zebras (except those for which this slot valuation is overridden) have *all* **zebra-color**s for their **color**.

In other words, the meaning of a category-valued slot in PARKA carries with it an implicit universal quantifier. There are cases for which an implicit

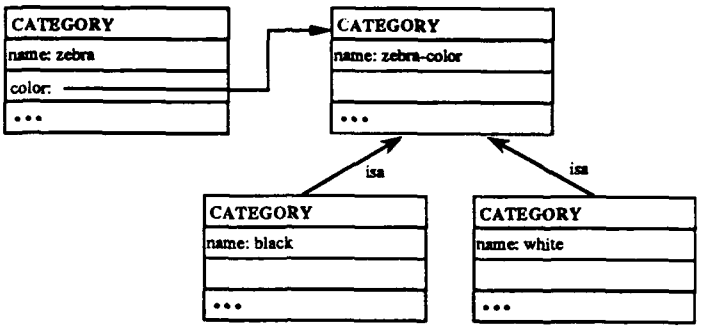

Figure 2: Correct use of a category-valued slot.

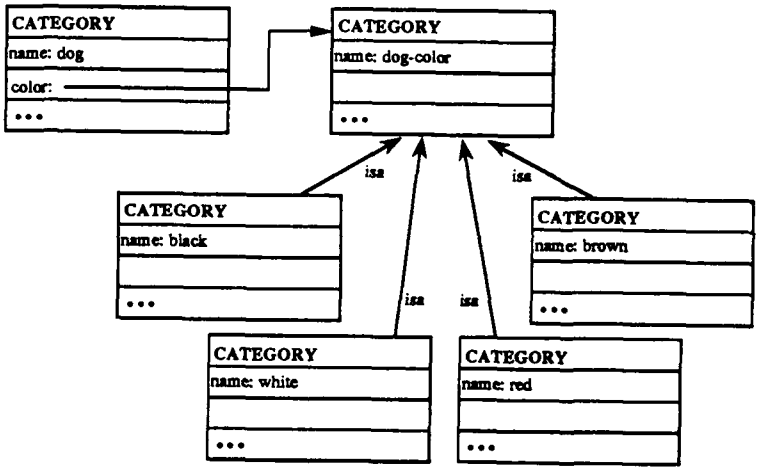

Figure 3: Incorrect use of a category-valued slot.

existential quantifier might seem more appropriate. For example, consider the **color** slot of the **dog** frame, and the **dog-color** category consisting of **black**, **white**, **red**, and **brown**. If we filled the **color** slot with **dog-color** (as in Figure 3) then we would be asserting that all dogs have *all* of these colors, which is clearly not correct. What we really want to say is that the colors of all dogs must be selected from the given category. Rather than tampering with the convention of universal quantification, PARKA provides an alternative mechanism for such cases.

It is often the case that we wish to describe a slot's filler as being *limited* to some set of values, without specifying any *particular* value for the filler in

any given case. In some previous systems this has been accomplished through the specification of predicates which valid fillers must satisfy (for example, [18]). PARKA's requirements for explicitness make this kind of specification undesirable; the use of a predicate representation would make it difficult to take advantage of PARKA's parallelism in responding to restriction-related queries or in checking the integrity of restrictions during updates. For this reason the admissible values for a given slot are represented *explicitly* by making the *type* of the slot RESTRICTION and by filling the slot with a category which is an ancestor of all and only those frames that are admissible fillers. This approach bears similarity to that taken in previous work (e.g., the role-set restrictions in [3]), and is analogous to the specification of *enumerated data types* in conventional programming languages. (This sort of comparison has been made elsewhere [8].)

A restriction is a constraint on the fillers of the given slot for *all* isa descendants of the frame at which the restriction is specified. If the type of the descendant's corresponding slot is also RESTRICTION then its filler must be a *sub-category* of the category specified in the original restriction. This sub-category then becomes a *sub-restriction* which further constrains all of the fillers of the given slot for frames further down in the isa hierarchy. If the descendant's corresponding slot is of the type SIMPLE or DEFINITIONAL then it must be filled with a *proper sub-category* or a *member* of the category specified in the restriction. The integrity of all restrictions is maintained at KB update time using an efficient parallel algorithm.

For example, returning to the "dog-color" problem, we would make the **color** slot of **dog** a restriction with value **dog-color** (as shown in Figure 4). Sub-categories of **dog** could provide sub-restrictions or simple values. For instance, the **color** of **Poodle** could be restricted to being just **black** or **white**, while the **color** of **Irish-Setter** could be set to **red**. Note that this would allow a black (mutant) Irish Setter but not a purple one (due to the restriction of the **color** slot of **dog**). Definitional values could be provided here as well, though it would be difficult to conjure up a case in which a dog's color is *definitional* with respect to some category term. The representations for individual dogs could also specify sub-restrictions to express incomplete information, or simple or definitional values in appropriate circumstances.

As another example of the utility of restrictions, consider a category **give-action** with three slots **giver**, **recipient**, and **object**. We could specify a re-

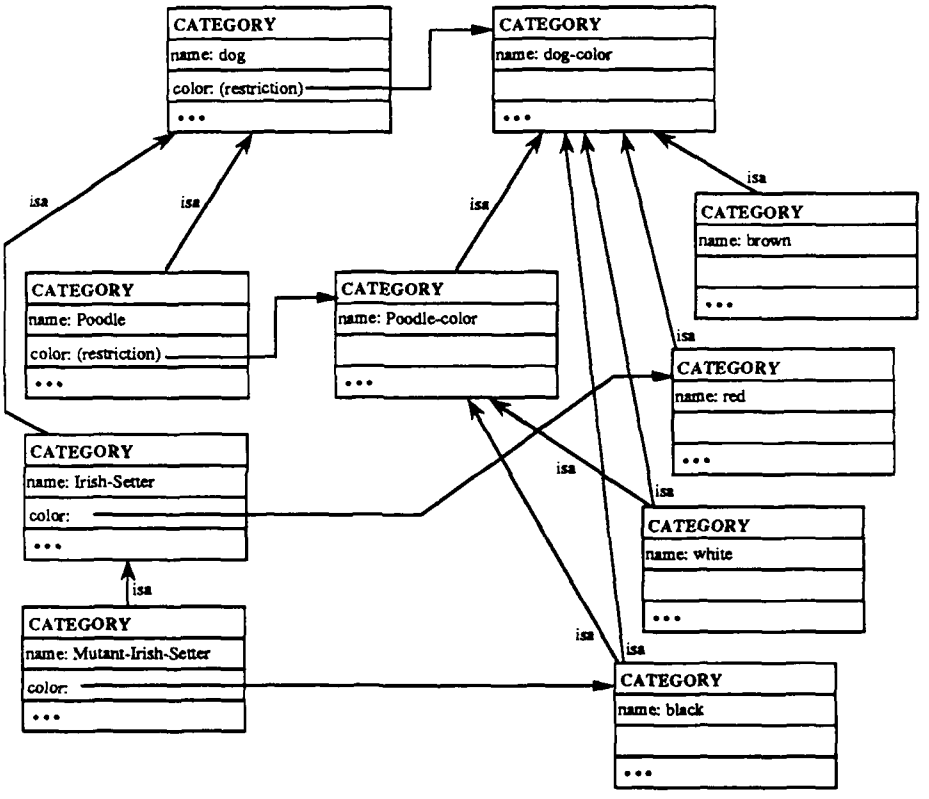

Figure 4: Restrictions on dog colors.

striction for the giver and recipient slots consisting of the category **animate-obj**. This would dissallow the creation of a give-action in which the giver was, say, an umbrella stand. More interestingly, we could create an isa child of give-action called **pay-action**. By specifying a restriction of the category **monetary-value** on the **object** slot of **pay-action** we dissallow the classification of non-monetary givings as pay-actions. Further, if no value is specified as the recipient of some individual pay-action, we will at least have the incomplete information that the recipient must be animate. We could create further specializations for payings by credit card, givings to charities, etc. Restrictions can thus be seen as providing mechanisms for object classification, for the representation of incomplete information, and also for the representation of action specialization.

There is a high degree of similarity between restrictions and definitional values. In fact, definitional values are really just restrictions to single-valued categories. We have kept definitional values and restrictions distinct mainly because definitional values seem to have a privileged epistemological status; although there is functional similarity between the two mechanisms it nonetheless seems natural to think of definitions and restrictions as different things. Further, we can optimize our implementation somewhat by singling out definitional values as a special case.

An additional issue regarding category-valued slots and restrictions involves *cardinality*. In some earlier systems it was possible to specify that a slot must be filled by a certain *number* of individuals, or that the number of individuals must fall within some given range. (These specifications have sometimes been called *cardinality facets* [8] [11].) This is clearly a desirable functionality but it is currently beyond PARKA's capabilities. If the cardinalities involved are small then we can in some cases simulate the desired functionality by using several slots; however, we are looking into the development of a more elegant solution for future versions of PARKA.

6 Set-Constructor Categories

Some previous frame systems provided the user with several inheritance methods and allowed the user to specify a different type of inheritance for each slot in a frame [11]. For example, 'Union' inheritance combined all of

the values for the given slot from *all* of the given frame's isa ancestors. Although this kind of specification appears to be very useful, PARKA allows only one kind of inheritance, and that kind of inheritance accesses at most one inherited value. Nonetheless, much of the functionality of 'union' inheritance, and more, can be captured in PARKA with a topology constraint mechanism called a *set-constructor category.*

PARKA has three kinds of set-constructor categories: *union, intersection,* and *set-difference.* To create a union the user specifies a set of *constituent categories* and a new name. The system then creates a new category that represents the union of the constituents. Initially this just means that the new frame must be made to be an isa parent of each of the constituents, and that all common parents of the constituents become isa ancestors of the new frame. A set-constructor category, however, is *self-maintaining.* This means that when later changes are made to the knowledge base, the system will always react with modifications to insure that the new frame is still an accurate representation of the set-theoretic union (or intersection or set-difference) of its constituents.

For example, suppose we create a union, **MotionPicture**, of categories **FilmMovie** and **Video**, and that we later add isa links to make some other category, **TemporalArtObject**, an isa parent of both **FilmMovie** and **Video**. Such a configuration of links, as shown in figure 5, means that **MotionPicture** is a subset of **TemporalArtObject**, and hence that an isa link should be added from **MotionPicture** to **TemporalArtObject**.

PARKA automatically checks all knowledge base updates to ensure that set-constructor categories are appropriately maintained. In particular, it must watch for additions and deletions of isa links, and must sometimes respond with the addition and/or deletion of other isa links.

Intersections and set-differences are similar to unions in functionality and in update maintenance requirements. An intersection is created by specifying a new name and a set of constituent categories. The new frame is maintained to always represent the set-theoretic intersection of the constituents, and will have the constituents as isa parents. A set-difference is created by specifying a *positive constituent* category and a *negative constituent* category. The set-difference is maintained to be the subset of the positive constituent which has no elements in the negative constituent. The set-difference frame is made

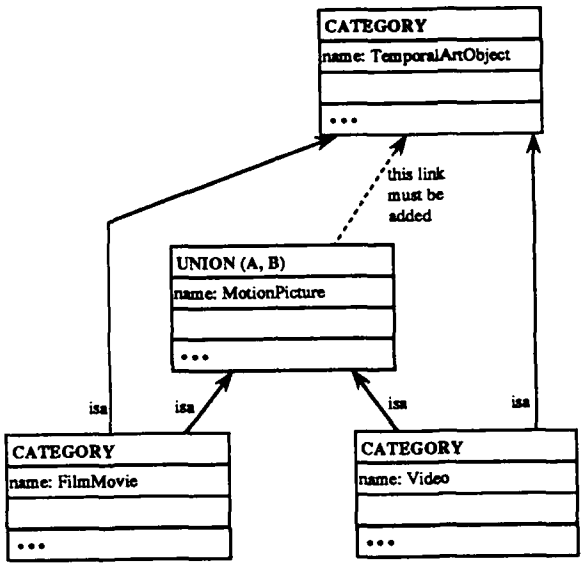

Figure 5: Maintaining a Union.

to be an isa child of the positive component; it has no isa connection to the negative component.

As an example of the utility of set-constructor categories, consider the representation of the **supporters** of **GeorgeBush** as the union of **Reagan-democrats** and **Republicans** who are not members of the ACLU. (See Figure 6.) If Joe Shmoe joins the ACLU then the system will automatically delete the isa link which describes him as a GOP hard-liner (and hence as a Bush supporter).

Although many parts of the update maintenance algorithms are highly parallelizable, and although most set-constructor categories need not even be checked for most updates, this maintenance is PARKA's least efficient aspect. One problem is that changes made to fix one set-constructor may mess up another one; i.e., the update maintenance procedure's updates need also to be checked and responded to. Another problem is that it is possible to place set-constructor categories at unwise positions in the hierarchy; this can cause a poorly placed category to be checked during every (or almost every) KB update. (For example, an intersection which is the root of the entire hierarchy will cause this problem.) This situation is somewhat ameliorated by turning

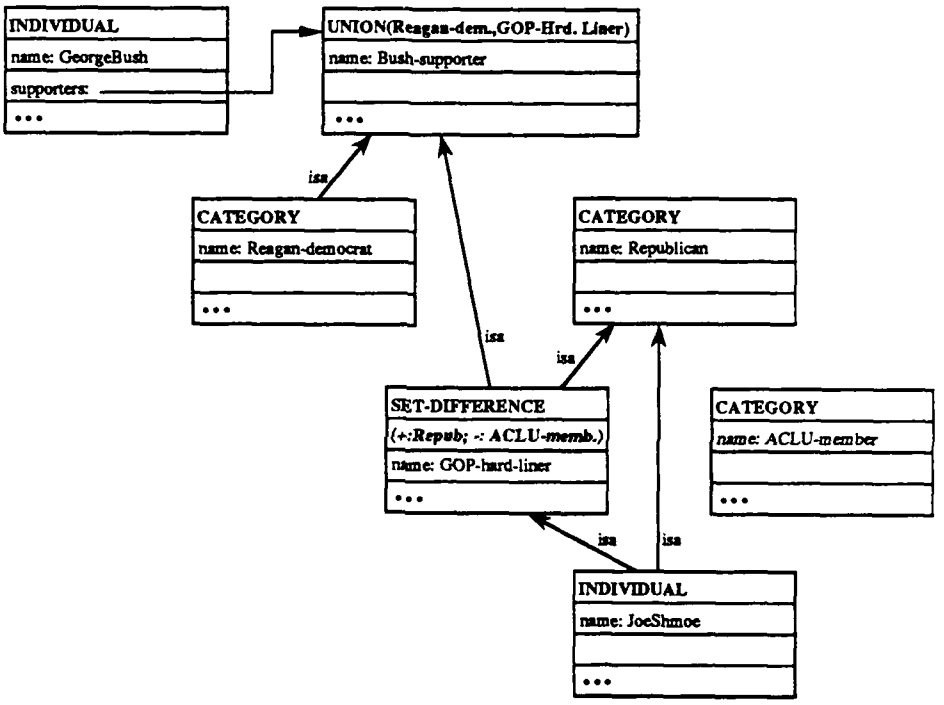

Figure 6: Set-constructor categories in action.

off all update maintenance while building a (presumably debugged) KB; all of the set-constructor categories can then be built in one 'batch' process.

We have still not fully analyzed all of the complexity issues involved in the maintenance of set-constructor categories. The problem is obviously related to that of computing subsumption relations amongst terms which are set-theoretically described. That problem, in various forms, has been shown by several authors to be intractable and in some cases undecidable ([9], [13], [15], [16]). Our problem is somewhat different, however, because (1) in our hierarchies all subsumption relations (except for the current update under consideration) are represented explicitly, (2) we are working on massively parallel hardware which allows us in some cases to trade space for time, and (3) we expect the number of set-constructor categories to be very small in comparison to the total number of frames in a PARKA KB. Further analysis of these issues is one of our short-term priorities.

7 Aggregations and Part/Whole Relations

PARKA's philosophy of using topology constraints, maintained at update-time, to represent knowledge-structuring information has been used to provide a mechanism for the principled and efficient representation of some part/whole relations. The representation of part/whole relations is a subfield of knowledge representation with considerable subtlety and a history of interesting, difficult problems (see, e.g. [10]). Winston et. al. have provided a classification for part/whole relations which yields six distinct types: 1. component-integral object (pedal-bike), 2. member-collection (ship-fleet), 3. portion-mass (slice-pie), 4. stuff-object (steel-car), 5. feature-activity (paying-shopping), and 6. place-area (Everglades-Florida) [23]. The *aggregation* mechanism in PARKA was developed specifically to deal with the component-integral object relation, although it may be capable of handling other part/whole relations (particularly feature-activity) as well.

Consider the knowledge base fragment depicted in Figure 7. We have used the **components** slot (which is of type RESTRICTION in all of the cases shown) to indicate the components of the various integral objects that are represented. The problem with this representation is that we expect part relations and isa relations to exhibit certain *transitivities* that are not explicit

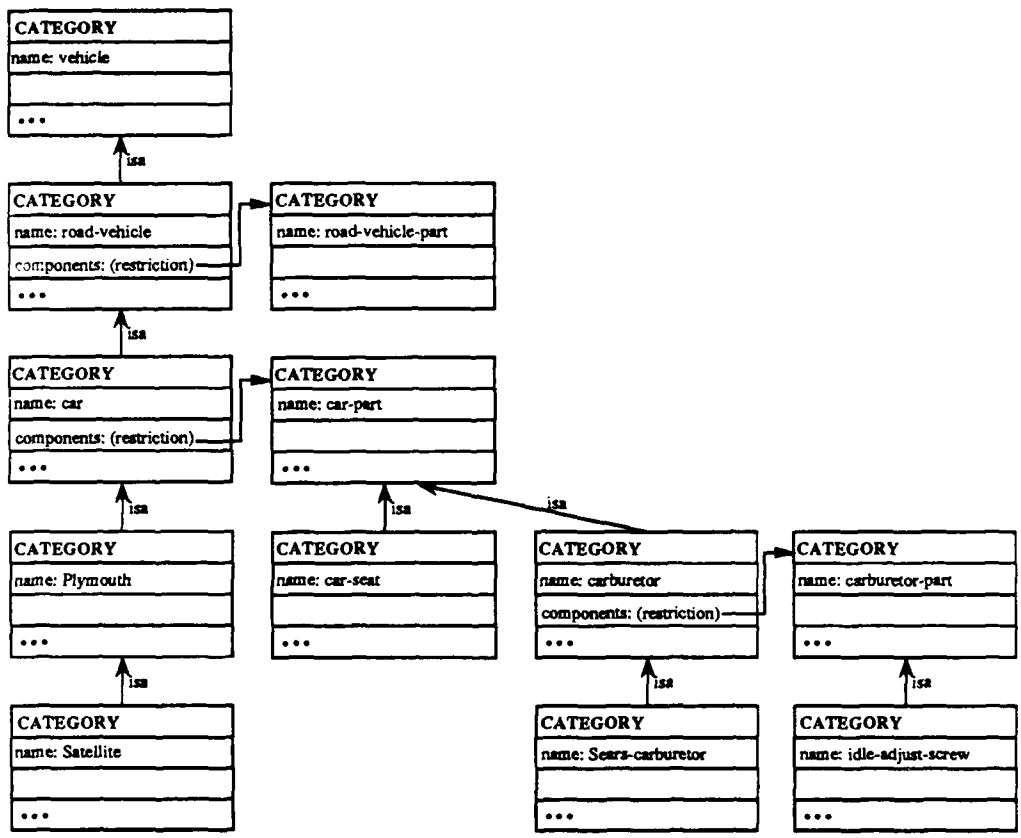

Figure 7: An incomplete part hierarchy.

in the figure. For example, we would like **idle-adjust-screw** to be a sub-category of **car-part**, and we would like **carburetor** to be a sub-category of **road-vehicle-part**. The situation can be rectified by adding two isa links - one from **car-part** to **road-vehicle-part**, and one from **carburetor-part** to **car-part**. However, we do not wish to force the knowledge base builder to make all such relations explicit. The user should be able to declare that the part frames in the figure should "behave correctly as components with respect to transitivity relations" and the system should make the details explicit; this is the function of aggregations.

An aggregation is created by providing the name of a slot, a new name for the aggregation frame, and a set of "auxiliary" frames. The existence of an

aggregation causes the enforcement (at update time) of a topology constraint which may be described as follows: the *fillers* of the given slot, for all isa descendants of the aggregation, must *themselves* be isa descendants of the aggregation. In addition, the *fillers* of the given slot, for all isa descendants of *all of the given auxiliary frames*, must also be isa descendants of the aggregation. Note that the constraint says only that the specified fillers must be isa *descendants* and not direct isa *children*; this allows PARKA to keep the number of added links to a minimum by computing upper-bound sets in the isa hierarchy.

In the given example we would create **road-vehicle-part** as an aggregation with slot **components** and auxiliary frame **road-vehicle**. (See Figure 8.) This would ensure that **car-part** becomes an isa descendant of **road-vehicle-part**. In addition, **car-part** would be an aggregation (with slot **components** and auxiliary frame **car**) so that **carburetor-part** would be made to be an isa descendant of **car-part**. (The PARKA code for this example is given in Appendix B.) Depending on the order in which these aggregations are created, there might also be a (redundant) isa link from **carburetor-part** to **road-vehicle-part**. The existence of such a redundant link cannot change the properties inherited by **carburetor-part**, however, because **car-part** will still be closer to **carburetor-part** by the inferential distance ordering.

In general, then, we can represent component-integral object relations (and perhaps other part/whole relations) by specifying the "parts" frame as an aggregation with the "whole" frame as an auxiliary frame and the part/whole relation-type (in this case **components**) as the slot. The desired transitivities will then be made explicit and maintained across updates by the system.

Although this mechanism goes a long way in solving part/whole representation problems, there are still (at least) two deficiencies. One problem is that a query for the value of the **components** slot of the **vehicle** frame in our example will return NO-VALUE. This problem can be handled by giving all root frames in the isa hierarchy **components** slots filled with appropriate aggregations, although we are exploring more elegant local solutions. A second and more subtle problem involves handling deletions of isa links in the wake of user-initiated changes. Returning to our example, if the **components** slot of **carburetor** is removed then we will no longer want **carburetor-part**

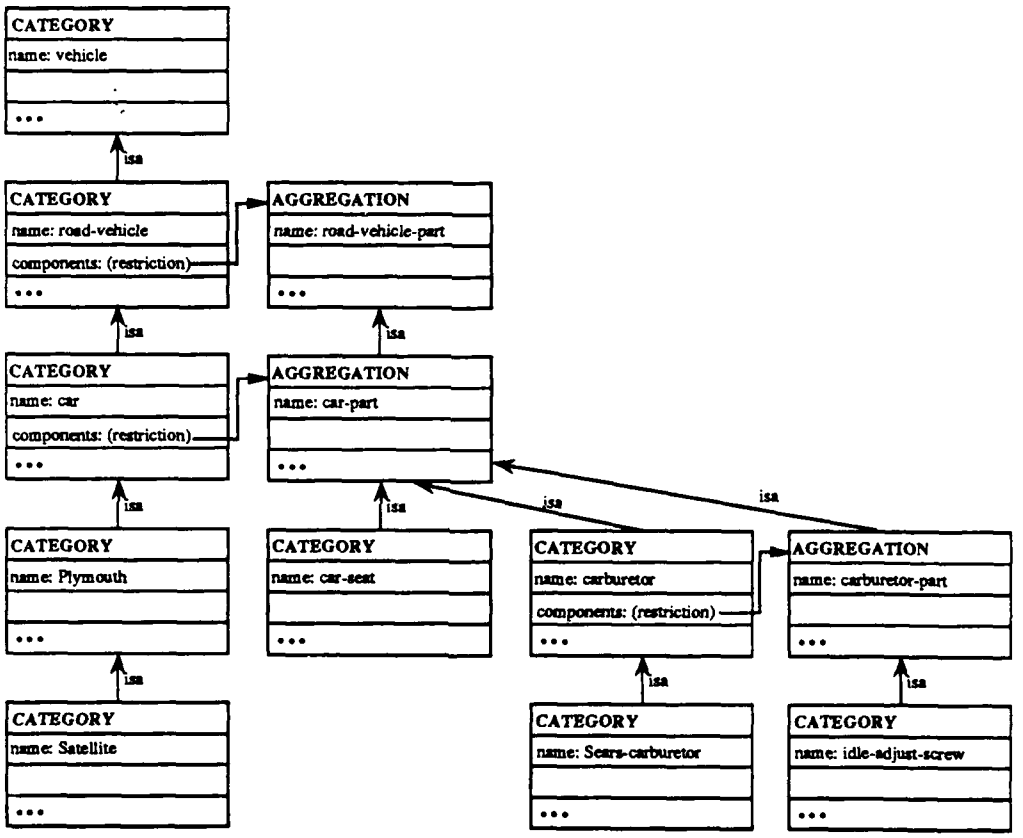

Figure 8: The corrected part hierarchy with aggregations.

to be an isa descendant of **car-part** (or of **road-vehicle-part**). However, if the *user* had *explicitly* created an isa link from, say, **idle-adjust-screw** to **car-part** then we would like for that link to remain unaffected by the above-mentioned change. In short, we want our topology constraints to be able to *undo* only those changes which they have themselves made. In the given case this can be accomplished by marking isa links as "user-created" or "system-created".

8 Conclusions

PARKA is primarily an experiment in the use of massive parallelism in symbolic knowledge representation. We have managed to capture much of the representational power of previous frame and semantic net systems, while realizing impressive gains in speed and efficiency, by replacing implicit or procedural knowledge with topology constraints maintained across updates. We have also managed to provide semantically sophisticated representational facilities within the PARKA framework, most notably in the treatment of multiple inheritance and in the mechanism for the representation of part/whole relations.

Nonetheless, PARKA still has several weaknesses. In comparison to other systems, the following capabilities (some of which have been discussed above) are noticeably absent: 1. There is no facility for the attachment of general procedures to frames (as in [1], [11], [5], [18]). 2. The support for the representation of numeric values (i.e., cardinality facets, numeric sub-ranges, etc.) is very weak (in contrast to, for example, [11]). 3. We have provided no conventions for the interpretation of sub-categorizations as *exhaustive* or *mutually exclusive*. 4. There is no easy way to represent relations amongst slots, or to create a taxonomy of the relations represented by slots (in contrast to [22] or to some extent [3]). 5. There is no mechanism for specifying the interdependency of restrictions on different slots of the same frame (again, see [3] and [22]).

Still, the capabilities that PARKA *does* have are at least as powerful as those of some of the "frame" systems that have been used in past AI work. PARKA builds such representational facilities into a very fast Connection Machine implementation and hence is unique among knowledge representa-

tion tools. Further, the PARKA project is still in its infancy and many of the above-mentioned deficiencies may be corrected in the future.

A The Syntax of the PARKA Language

The following gives the syntax of PARKA's most important user functions.

```
;; CONSTANTS

(defconstant *NO-VALUE* '*NO-VALUE*
  "*NO-VALUE* is used in PARKA to indicate that a slot
HAS NO VALUE, as opposed to having the value nil.  The
symbol *NO-VALUE* evaluates to itself.")

;; BASIC FRAME CREATION, DELETION, PRINTING, AND ACCESS FUNCTIONS

(defun init-PARKA ()
   "Clears the PARKA knowledge base." ...)

(defun make-individual (name)
   "Creates an individual frame with the given name
and adds it to the PARKA knowledge base." ...)

(defun make-category (name)
   "Creates a category frame with the given name
and adds it to the PARKA knowledge base." ...)

(defmacro f (name)
   "Returns the frame structure that has the given name as
its frame-name." ...)

;; reader macros define an alternative (preferred)
;; syntax for user-entry of frames using square brackets:
;; [foo] = (f foo) => the frame named foo

(defun names-a-frame (n)
   "Returns t if the given name is the name of a frame in the
current PARKA knowledge base." ...)

(defun frame-p (f)
   "Returns t if f is a frame; returns nil otherwise." ...)
```

```
(defun individual-p (frame)
   "Returns t if the given frame is an individual." ...)

(defun category-p (frame)
   "Returns t if the given frame is a category" ...)

(defun delete-frame (f)
   "Deletes a frame and all references to it in other frames.
The last deleted frame may be restored with restore-last-frame.
Delete-frame always returns t." ...)

(defun restore-last-frame ()
   "Restores the last deleted frame (and all references to it)
to the knowledge base.  Restore-last-frame returns t unless
there is no frame to be restored - in that case nil is returned." ...)

(defun print-frame (f &optional stream level)
   "Prints a frame structure in a readable form.  This is
the :print-function for frame structures.  It prints only
those slot values that are explicit at the given frame." ...)

(defun print-all-frames ()
   "Prints all frames in the PARKA knowledge base in a
readable form."  ...)

;; FUNCTIONS FOR ISA LINK CREATION DELETION, ETC.

(defun isa (frame category)
   "Creates an isa link from the first argument to the second.
If the arguments are not of the proper type an error is signaled.
If the arguments are of the proper type then isa always returns t." ...)

(defun delete-isa (frame category)
   "Deletes any existing isa link from the first argument to the
second.  Delete-isa always returns t." ...)

(defun children (frame)
   "Returns a list of all of the (immediate) isa children
of the given frame" ...)

(defun parents (frame)
   "Returns a list of all of the (immediate) isa parents
of the given frame" ...)
```

```
(defun child-p (f1 f2)
   "Returns t if frame f1 is an (immediate) isa child of
frame f2" ...)

(defun parent-p (f1 f2)
   "Returns t if frame f1 is an (immediate) isa parent of
frame f2" ...)

(defun descendants (frame)
  "Returns a list of all isa descendants of the given frame." ...)

(defun ancestors (frame)
  "Returns a list of all isa ancestors of the given frame." ...)

(defun descendant (f1 f2)
   "Returns t if frame f1 is an isa descendant of frame
f2.  Returns nil otherwise." ...)

(defun ancestor (f1 f2)
   "Returns t if frame f1 is an isa ancestor of frame
f2.  Returns nil otherwise." ...)

; FUNCTIONS FOR SLOT FILLING AND UNFILLING, VALUE ACCESS, ETC.

(defun fill-slot (frame slot value &optional slot-type)
   "Explicitly fills the given slot of the given frame
with the given value, and makes that slot to be of
type slot-type.  Any previous value for the given
slot is lost.  Fill-slot always returns t."  ...)

(defun unfill-slot (frame slot)
  "If the given frame is explicitly valued for the given slot
then this deletes the explicit valuation; otherwise it has no
effect.  Returns t if a valuation was deleted and nil otherwise." ...)

(defun xval-p (frame slot)
  "Returns t if the given frame has an explicit value for
the given slot, and nil otherwise.  In this context restrictions
do not count as explicit values." ...)

(defun xrest-p (frame slot)
  "Returns t if the given frame has an explicit RESTRICTION for
the given slot, and nil otherwise." ...)
```

```
(defun xrest (frame slot)
   "If the given slot has an explicit restriction then that
restriction is returned; otherwise the value is nil." ...)

(defun restrictions (frame slot)
   "Returns a list of restrictions which apply to the given
slot for the given frame.  The value returned is a list of the closest
explicit restriction slot fillers on each upward isa path.  If the given
frame has an explicit restriction then a singleton list of that
restriction is returned." ...)

(defun meets-restrictions (frame slot val)
  "Returns t if val is an isa descendant of all restrictions of slot
for frame." ...)

(defun definitional (frame slot)
   "Returns t if the given frame itself has, or is
an isa descendant of a frame that has, a definitional
value for the given slot." ...)

(defun definitional-val (frame slot)
   "If the given frame itself has a definitional value for the
given slot, or if it is an isa descendant of a frame that has a
definitional value for the given slot, then this returns that
definitional value." ...)

;; TOP LEVEL SLOT-VALUE QUERIES

(defun slot-value (frame slot)
  "Returns the value of the given slot for the given frame.
If the given frame is not explicitly valued for the given slot
then an inherited value is computed according to the inferential
distance ordering.  Note that Restriction values are ignored in
this computation." ...)

(defun frames-with (slot value)
  "Returns a list of all frames in the system which have - explicitly
or inherited - the given value for the given slot." ...)

(defun frames-with-some (slot value)
   "Returns a list of all frames in the system which have a value
(explicit or inherited) for the given slot which is a SUPER-CATEGORY
of the given value." ...)
```

```
(defun frames-with-all (slot value)
   "Returns a list of all frames in the system which have a value
(explicit or inherited) for the given slot which is a SUB-CATEGORY
of the given value." ...)

;; SET-CONSTRUCTOR FRAME CREATION AND ACCESS FUNCTIONS

(defun make-intersection (name &rest constituents)
   "Creates a frame with the given name that will be maintained
across updates to always represent the set-theoretic intersection of
the frames provided as constituents." ...)

(defun intersection-p (f)
   "Returns t if f is an intersection frame; returns nil otherwise." ...)

(defun all-intersections ()
   "Returns a list of all intersection frames in the PARKA knowledge
base." ...)

(defun make-union (name &rest constituents)
   "Creates a frame with the given name that will be maintained
across updates to always represent the set-theoretic union of
the frames provided as constituents." ...)

(defun union-p (f)
   "Returns t if f is a union frame; returns nil otherwise." ...)

(defun all-unions ()
   "Returns a list of all union frames in the PARKA knowledge
base." ...)

(defun make-set-difference (name pos neg)
   "Creates a frame with the given name that will be maintained
across updates to always represent the set-theoretic set difference
of the pos and neg categories" ...)

(defun set-difference-p (f)
   "Returns t if f is a set-difference; returns nil otherwise." ...)

(defun all-set-differences ()
   "Returns a list of all set-difference frames in the PARKA knowledge
base." ...)

;; AGGREGATION FRAME CREATION AND ACCESS FUNCTIONS
```

```
(defun make-aggregation (name ag-slot &rest aux-frames)
   "Creates a frame with the given name which maintains an aggregation
topology constraint with respect to the given ag-slot and the given
aux-frames." ...)

(defun aggregation-p (f)
   "Returns t if f is an aggregation frame; returns nil otherwise." ...)

(defun all-aggregations ()
   "Returns a list of aggregation frames in the PARKA knowledge
base." ...)

;; BATCH UPDATE FACILITY

(defun batch-update (path)
   "BATCH-UPDATE should be used whenever the user wishes to create a large
knowledge base from a file of definition forms.  BATCH-UPDATE loads the
file specified in the path argument with no update checking, and then
attempts to satisfy all topology constraints.  This is considerably
faster than executing a long sequence of ... al (update-checked) updates."
 ...)

;;; USER OPTION

(defvar *update-verbose* t
   "If *update-verbose* is non-nil then the update procedures will
send notifications of various actions to standard output.")
```

B Code for the Car Parts Example

The following code creates the structures described in the Aggregations section and diagramed in figure 8.

```
(make-category 'vehicle)
(make-category 'road-vehicle)
(isa [road-vehicle][vehicle])
(make-category 'car)
(isa [car][road-vehicle])
(make-category 'plymouth)
(isa [plymouth][car])
(make-category 'satellite)
```

```
(isa [satellite][plymouth])
(make-aggregation 'road-vehicle-part 'components [road-vehicle])
(fill-slot [road-vehicle] 'components [road-vehicle-part] 'restriction)
(make-aggregation 'car-part 'components [car])
(fill-slot [car] 'components [car-part] 'restriction)
(make-category 'seat)
(isa [seat][car-part])
(make-category 'carburetor)
(isa [carburetor] [car-part])
(make-category 'sears-carb)
(isa [sears-carb] [carburetor])
(make-aggregation 'carb-part 'components [carburetor])
(fill-slot [carburetor] 'components [carb-part] 'restriction)
(make-category 'idle-adjust-screw)
(isa [idle-adjust-screw][carb-part])
```

References

[1] Bobrow, Daniel G., and Winograd, Terry, "An Overview of KRL, a Knowledge Representation Language," *Cognitive Science* **1** (1), 1977, 3-46.

[2] Brachman, Ronald J., Fikes, Richard E., and Levesque, Hector J., "KRYPTON: a Functional Approach to Knowledge Representation," FLAIR Technical Report No. 16, Fairchild Laboratory for Artificial Intelligence Research, Palo Alto, CA, May, 1983.

[3] Brachman, Ronald J., and Schmolze, James G., "An Overview of the KL-ONE Knowledge Representation System," *Cognitive Science* **9**, 1985, 171-216.

[4] Brachman, Ronald J., ""I Lied about the Trees," Or, Defaults and Definitions in Knowledge Representation," *AI Magazine*, Fall, 1985, 80-93.

[5] Charniak, Eugene, Reisbeck, Christopher K., McDermott, Drew V., Meehan, James R., *Artificial Intelligence Programming* Lawrence Erlbaum Associates, Publishers, New Jersey, 1987.

[6] Evett, Matthew, Spector, Lee, and Hendler, James, "Knowledge Representation on the Connection Machine," *Supercomputing 89: Proceedings of the Conference,* Reno, Nevada, 1989, published by ACM, New York, New York, 1989.

[7] Evett, Matthew, Spector, Lee, and Hendler, James, "PARKA: A Symbolic Knowledge Representation System on the Connection Machine," (forthcoming) Technical Report, Department of Computer Science, University of Maryland, College Park, MD, 1990.

[8] Fikes, Richard, and Kehler, Tom, "The Role of Frame-Based Representation in Reasoning," *Communications of the ACM,* September 1985, Volume 28, Number 9, 904-920.

[9] Haase, Kenneth W. Jr., "TYPICAL - A Knowledge Representation System," MIT Artificial Intelligence Laboratory Technical Report 988, 1987.

[10] Hayes, Patrick J., "Some Problems and Non-Problems in Representation Theory," *Proc. AISB Summer Conference,* University of Sussex, 1974, 63-79.

[11] IntelliCorp, *IntelliCorp KEE Software Development System User's Manual,* 1985.

[12] Kripke, Saul A., *Meaning and Necessity,* Harvard University Press, Cambridge, Massachusetts, 1972.

[13] Levesque, Hector J., and Brachman, Ronald J., "A Fundamental Tradeoff in Knowledge Representation and Reasoning (Revised Version)," in *Readings in Knowledge Representation,* Ronald J. Brachman and Hector J. Levesque, editors, Morgan Kaufmann Publishers, Inc., 1985.

[14] Minsky, M., "A Framework for Representing Knowledge," in *Mind Design,* J. Haugeland, editor, The MIT Press, Cambridge, MA, 1981, 95-128.

[15] Nebel, Bernhard, "Computational Complexity of Terminological Reasoning in BACK," in *Artificial Intelligence* 34 (1988), 371-383, 1988.

[16] Patel-Schneider, Peter F., "Undecidability of Subsumption in NIKL," *Artificial Intelligence* 39 (1989), 263-272.

[17] Putnam, Hilary, "The Meaning of 'Meaning,' " in *Language, Mind, and Knowledge,* K. Gunderson, editor, Minnesota Studies in the Philosophy of Science, 7, University of Minnesota Press, 1975.

[18] Roberts, Bruce R., and Goldstein, Ira P., "The FRL Manual," Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 409, September 1977.

[19] Shastri, Lokendra, *Semantic Networks: An Evidential Formalization and its Connectionist Realization,* Morgan Kaufman, 1988.

[20] Stefik, Mark, and Bobrow, Daniel G., "Object-Oriented Programming: Themes and Variations," in *AI Magazine*, Winter, 1986, 40-62.

[21] Touretzky, D.S., *The Mathematics of Inheritance Systems,* Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.

[22] Wilensky, Robert, "Some Problems and Proposals for Knowledge Representation," Report No. UCB/CSD 86/294, University of California, Berkeley, May 1986.

[23] Winston, Morton E., Chaffin, Roger, and Herrmann, Douglas, "A Taxonomy of Part-Whole Relations," *Cognitive Science* **11**, 417-444, 1987.

[24] Woods, William A., "What's in a link: Foundations for Semantic Networks," in *Representation and Understanding: Studies in Cognitive Science,* D. G. Bobrow and A. M. Collins, editors, Academic Press, New York, 1975, 35-82.